Philosophia Scientiæ
Travaux d'histoire et de philosophie des sciences

17-3 | 2013
Tacit and Explicit Knowledge: Harry Collins's Framework

Unix selon l'ordre des raisons : la philosophie de la pratique informatique

Baptiste Mélès

Édition électronique
URL : http://journals.openedition.org/philosophiascientiae/897
DOI : 10.4000/philosophiascientiae.897
ISSN : 1775-4283

Éditeur
Éditions Kimé

Édition imprimée
Date de publication : 1 octobre 2013
Pagination : 181-198
ISBN : 978-2-84174-641-5
ISSN : 1281-2463

Référence électronique
Baptiste Mélès, « Unix selon l'ordre des raisons : la philosophie de la pratique informatique », *Philosophia Scientiæ* [En ligne], 17-3 | 2013, mis en ligne le 01 octobre 2016, consulté le 05 novembre 2020. URL : http://journals.openedition.org/philosophiascientiae/897 ; DOI : https://doi.org/10.4000/philosophiascientiae.897

Unix selon l'ordre des raisons : la philosophie de la pratique informatique

Baptiste Mélès[*]

Laboratoire d'Histoire des Sciences et de Philosophie,
Archives H. Poincaré (UMR 7117),
CNRS, Université de Lorraine, Nancy (France)

Résumé : Il est parfois fécond, en philosophie des sciences, de chercher si les concepts techniques relèvent d'une nécessité de structure plutôt que des seuls hasards de l'invention. En essayant de fonder de la sorte les concepts fondamentaux des systèmes d'exploitation que sont les notions de processus et de fichier, on s'aperçoit qu'ils sont, depuis Unix, les pendants des notions ontologiques abstraites d'acte et d'objet, et qu'ils satisfont toutes les propriétés que la théorie des catégories peut en attendre. La programmation peut dès lors être vue comme une thématisation, c'est-à-dire la transformation de l'acte qu'est le processus en l'objet qu'est le fichier ; mais on découvre également que l'exécution de programmes joue le rôle d'une « antithématisation », transformation d'objets en actes dont l'histoire récente des mathématiques fournit également des exemples. Par ses concepts comme par ses procédés, la pratique informatique est pour la philosophie un objet aussi pur que les mathématiques.

Abstract: In philosophy of science, it is sometimes fruitful to examine whether technical concepts originate from a structural necessity, rather than from the contingency of invention. Operating systems, for instance, rely on two main concepts: the notions of process and file. Trying to identify their raison d'être, we can see that they are, since Unix, the transposition in computer science of the abstract ontological notions of act and object, and that they satisfy all the properties that category theory may expect. Programming can therefore be seen as a "thematization," i.e. the transformation of the act or process into an object or file. But we also realize that the execution of programs acts as an "antithematization:" a transformation of objects into acts, which also has instances in recent history of mathematics. Concepts and methods of computing practices can be for philosophy as pure objects as mathematics are.

∗. L'auteur remercie les relecteurs anonymes pour leurs précieuses remarques.

1 Introduction : la philosophie de la pratique informatique

La méthode est parfois féconde en philosophie des sciences, de chercher si les concepts techniques relèvent d'une nécessité conceptuelle ou des hasards de l'invention.

Telle était déjà la gageure de la *Philosophie de la nature* de Hegel [Hegel 2004], qui rapportait à leur construction logique l'ensemble des concepts par lesquels l'esprit appréhende ce qui lui paraît au premier abord le plus extérieur et le plus étranger : la nature. Ce projet n'était peut-être pas si stérile qu'on a voulu le croire. Jean Cavaillès [Cavaillès 2008a], puis le Jules Vuillemin de la *Philosophie de l'algèbre* [Vuillemin 1962], se sont efforcés, en mettant au jour le contenu philosophique immanent aux concepts scientifiques, de comprendre quelle était la part de nécessité interne — c'est-à-dire dans quelle mesure la structure de la connaissance elle-même justifiait le cheminement de l'invention — dans le développement d'une science en un moment charnière de l'histoire des mathématiques.

La méthode est aisément transposable dans l'encore jeune philosophie de l'informatique. L'idée même d'une philosophie de l'informatique est parfois encore accueillie avec surprise ; mais si l'on est prêt à concéder que les *concepts* mathématiques présentent quelquefois de quoi piquer l'intérêt du philosophe, que les *pratiques* et les *œuvres* des mathématiciens ne sont pas moins passibles d'un examen philosophique que leurs concepts, et si l'on est prêt à appeler cela une philosophie des mathématiques, nous espérons montrer que les concepts, les pratiques et les œuvres des informaticiens peuvent faire l'objet d'une philosophie de l'informatique tout aussi instructive.

Peut-être au passage délivrera-t-on les sciences de l'ingénieur en général de certain préjugé : de ce qu'elles sont applicables, on croit pouvoir les conclure empiriques. Mais à qui prend la peine de les examiner, elles n'apparaissent guère moins belles que les mathématiques pures : bien loin que l'esprit s'y soumette à la matière, c'est la matière qui n'y existe qu'à titre de construction de l'esprit ; leur monde est celui de Berkeley. La philosophie de l'informatique en particulier n'est donc pas celle d'une technique sans âme — à supposer qu'il en existe.

Il n'est d'ailleurs nul besoin de se borner aux abstractions de l'informatique théorique pour montrer que l'informatique n'a pas les mains moins propres que les mathématiques pures : la *philosophie de la pratique informatique*, c'est-à-dire l'étude de la *pratique* qu'est la programmation et des *œuvres* que sont les programmes, n'est pas moins instructive que la *philosophie de l'informatique théorique* développée par les logiciens. Nous essaierons de présenter ici une application de cette méthode en étudiant si l'on peut fonder en raison les deux notions fondamentales des systèmes d'exploitation que sont les concepts de fichier et de processus.

2 Une ontologie d'actes et d'objets

Les programmes que nous utilisons au quotidien — traitements de texte, navigateurs, etc. — ne sont pas écrits dans les langages abstraits des théoriciens de l'informatique tels le lambda-calcul ou les machines de Turing, mais dans ce que l'on pourrait appeler des langages « naturels » de programmation comme le C et le C++. Or tous ces programmes reposent en dernière instance sur un programme fondamental, qui administre directement le matériel informatique — processeur, mémoire vive, disques durs, etc. — pour le rendre accessible à l'utilisateur humain : le *système d'exploitation*. Le système d'exploitation est en ceci le programme des programmes.

Or toute étude des systèmes d'exploitation commence par une exposition des concepts de fichier et de processus. C'est ainsi que Andrew Tanenbaum, créateur en 1987 du système d'exploitation MINIX, dérivé simplifié d'Unix, écrit dans la deuxième édition d'*Operating Systems* : « Les appels système MINIX se répartissent en gros en deux grandes catégories : ceux qui se rapportent à des processus et ceux qui se rapportent au système de fichiers [Tanenbaum & Woodhull 1997, 15][2] ». L'auteur entreprend ensuite de « passer en revue » ces concepts, l'essentiel étant pour lui, et pour ses étudiants informaticiens, d'en venir le plus vite possible aux problèmes et solutions concrets qui en dérivent — système de fichiers, arborescence, droits, lecture et écriture des fichiers, hiérarchie et priorité des processus, états, signaux, fils d'exécution, ordonnancement, parallélisme. Dans le cadre de l'exposé technique, il suffit que les notions semblent tomber du ciel, sans que l'auteur ait à déterminer si elles ont été dictées par Dieu, mathématiquement démontrées, sélectionnées pour leur efficacité pratique, ou simplement sédimentées par la tradition ; il suffit d'introduire une notion nouvelle par la formule « Un autre concept clé présent dans quasiment tous les systèmes d'exploitation est... » [Tanenbaum 2003, 41].

Mais ces formules ont des parfums d'empirisme qui ne sauraient satisfaire le philosophe. On aimerait savoir pourquoi l'exposition des systèmes d'exploitation passe par ces notions précises, si cela a toujours été le cas et le sera toujours. Pourquoi le processus et le fichier sont-ils donc les notions fondamentales de ces programmes, eux-mêmes fondamentaux, que sont les systèmes d'exploitation ?

2.1 Les processus comme actes

Qu'est-ce d'abord qu'un processus ? Le terme est souvent défini comme « l'instance d'exécution d'un programme » [Tanenbaum & Woodhull 1997, 15], [Bovet & Cesato 2001, 10], [Tanenbaum 2003, 37], [Lions 1996, 6-3c]. Mais la définition est philosophiquement peu éclairante en ce qu'elle présuppose

2. L'expression « en gros » de Tanenbaum cache en réalité d'autres formes d'appels système, qui se rapportent par exemple au temps, comme les appels `gettimeofday` et `select`.

la connaissance de ce qu'est un programme ; et techniquement inexacte, car même si ce cas est le plus fréquent, rien n'exige que le processus dérive d'un programme écrit à l'avance.

John Lions — auteur en 1976 d'un commentaire du code source de la version 6 d'Unix qui fut aussitôt reconnu comme classique, et qui a longtemps circulé sous le manteau avant d'être enfin distribué officiellement [Lions 1996, 3] — a bien vu que le problème de définition du processus dépassait la simple technique et relevait de la philosophie :

> Fournir une définition généralement acceptable de la notion de « processus » soulève de nombreuses et de sérieuses difficultés. Elles s'apparentent à celles qu'affronte le philosophe voulant répondre à la question « qu'est-ce que la vie ? ». Nous serons loin d'être les seuls à mettre légèrement de côté les points les plus subtils. [Lions 1996, 7-1a]

C'est donc précisément ici que le philosophe prend le relai de l'informaticien : il lui faut cueillir les concepts à la racine.

L'hygiène la plus élémentaire en philosophie de la pratique informatique requiert d'adopter le principe méthodologique suivant : *l'essence des notions réside tout entière dans leur code source*. La réponse à la question de savoir ce qu'est un processus est donc à chercher dans la table des descripteurs de processus, qui résume l'ensemble des propriétés essentielles que le système d'exploitation a besoin de connaître et de manipuler.

Le code source sur lequel nous nous appuierons principalement est celui de la version 6 d'Unix (1975). Pourquoi ce système en particulier ? D'abord, parce que son code source a été intégralement publié ; ensuite, parce qu'il comprend toutes les caractéristiques d'un système Unix mature ; enfin, parce que sa brièveté exceptionnelle — 9000 lignes — lui permet d'être appréhendé dans sa totalité par un seul être humain. Nous avons donc la chance rare de pouvoir étudier un système libre, complet et minimal[3]. Le premier critère exclut les systèmes propriétaires tels que Microsoft Windows et MacOS ; le deuxième, les toutes premières versions d'Unix ; le troisième, les systèmes libres nés dans les années 1990 comme GNU/Hurd (commencé en 1990), Linux (1991) et les systèmes BSD récents (FreeBSD, NetBSD et OpenBSD, 1993–1995). On aurait en revanche tout aussi bien pu étudier MINIX, dont le code source n'est que trois fois plus long que celui de son ancêtre Unix.

Examinons donc la structure de données que cette version d'Unix décrit en langage C dans le fichier d'en-tête `proc.h` [Lions 1996, 7-2] :

3. Le critère de minimalité a des conséquences que l'on ne saurait négliger ; il conduit par exemple à réviser les rapports entre les processus et les fils d'exécution (*threads*), à mettre de côté les innovations liées au matériel et notamment au parallélisme. C'est le prix à payer pour une première approche, qui devra donc être complétée par une analyse des dynamiques à l'œuvre dans l'histoire des systèmes d'exploitation — analyse qui, rompant avec le critère de minimalité, devra s'appuyer sur un corpus plus vaste et plus complexe.

```
0358: struct proc
0359: {
0360:   char p_stat;
0361:   char p_flag;
0362:   char p_pri;    /* priority, negative is high */
0363:   char p_sig;    /* signal number sent to this process */
0364:   char p_uid;    /* user id, used to direct tty signals */
0365:   char p_time;   /* resident time for scheduling */
0366:   char p_cpu;    /* cpu usage for scheduling */
0367:   char p_nice;   /* nice for scheduling */
0368:   int  p_ttyp;   /* controlling tty */
0369:   int  p_pid;    /* unique process id */
0370:   int  p_ppid;   /* process id of parent */
0371:   int  p_addr;   /* address of swappable image */
0372:   int  p_size;   /* size of swappable image (*64 bytes) */
0373:   int  p_wchan ; /* event process is awaiting */
0374:   int  *p_textp; /* pointer to text structure */
0375:
0376: } proc[NPROC];
```

Langage C

Un processus y apparaît comme une structure (`struct proc`), c'est-à-dire un « faisceau » de données composé en particulier d'un identifiant numérique unique ou PID (`p_pid`), de l'identifiant d'un certain utilisateur du système (`p_uid`), d'un certain état (`p_stat` — ces états, définis aux lignes 382–387, déterminent notamment si le processus est en cours de création, d'exécution, de sommeil, ou de terminaison), d'un certain niveau de priorité (`p_pri`), de certains fanions (`p_flag`) qui sont des propriétés binaires (les différents fanions sont définis aux lignes 391–396), etc. La structure de données `proc` définit ainsi quelles sont les *propriétés* élémentaires d'un processus ; celles-ci se sont certes considérablement enrichies depuis Unix 6, comme on peut le constater en comparant sa taille croissante dans les codes sources de MINIX 2.0 [Tanenbaum & Woodhull 1997, 580–581], Linux 2.2 [Bovet & Cesato 2001, 65–66], et la taille qu'elle occupe actuellement dans Linux 3.2 ; mais toutes les propriétés originelles sont demeurées intactes.

A-t-on tout dit du processus quand on a décrit la structure de données qui lui est associée ? Certes non : en informatique comme en ontologie, le tout n'est pas de savoir ce que sont les choses ; il faut savoir si elles existent, et ce qu'elles peuvent devenir. Aussi faut-il examiner, comme le fait méthodiquement la théorie de la démonstration depuis Gentzen [Gentzen 1955], et comme le proposent dans un autre contexte Raymond Turner et Amnon Eden [Turner & Eden 2007], comment ces objets sont d'abord *introduits* et ensuite *éliminés*. Tels sont les actes canoniquement associés aux objets.

Un processus standard est toujours introduit, sous Unix, par un appel à la fonction `newproc()` (commentée dans Lions [1996, 6-4a et 4-5]), définie dans le fichier `slp.c` (lignes 1826–1919). Cette fonction ajoute un nouvel élément à la table des descripteurs de processus, le munit d'un nouvel identifiant (`p_pid`), et enregistre l'identifiant du processus qui l'a créé (`p_ppid`). Le processus est

donc un acte engendré par un acte antérieur, selon une chaîne — ou plus précisément un arbre — de mises en branle.

Tout naturellement se pose la question du premier moteur. Le processus `init` joue pour ainsi dire le rôle de la *causa sui*, voire d'une création continuée. Dans Unix 6, la structure de processus est initialisée — c'est-à-dire passe de l'essence à l'existence — dans la fonction `main()` du fichier `main.c`, qui est la partie principale du programme :

```
1589: proc[0].p_addr = *ka6;
1590: proc[0].p_size = USIZE;
1591: proc[0].p_stat = SRUN;
1592: proc[0].p_flag =| SLOAD|SSYS;
```

Langage C

Cette initialisation (commentée dans Lions [1996, 6-3c–6-4a]) détermine d'abord l'adresse (`p_addr`) et la taille (`p_size`) du segment de données associé au processus 0 ; elle attribue ensuite à ce processus l'état `SRUN`, c'est-à-dire « prêt à être exécuté » (*ready to run*) ; elle indique enfin par les fanions (*flags*) `SLOAD` et `SSYS` que le processus initial réside et résidera toujours dans la mémoire centrale. Dans Linux 2.2,

> l'ancêtre de tous les processus, appelé le *processus 0* [...], est un thread noyau créé à partir de rien durant la phase d'initialisation de Linux par la fonction `start_kernel()` [...]. Le thread noyau créé par le processus 0 exécute la fonction `init()` [...]. Puis `init()` invoque l'appel système `execve()` pour charger le programme *init*. [...] Le processus *init* ne se termine jamais, car il crée et surveille l'activité de tous les processus qui implémentent les couches externes du système d'exploitation. [Bovet & Cesato 2001, 93–95]

Les processus sont ainsi « créés », ou plutôt mis en branle, par d'autres processus, dont l'ancêtre ultime est un premier moteur créé *ex nihilo*, créateur et maître de lui-même et de toutes choses. Contingents comme le sont les actes ici-bas, ils sont « détruits » par un appel système ou en recevant un certain signal envoyé par un processus habilité à le faire : ainsi le Créateur nous rappelle-t-il à lui. Les processus sont donc des actes possédant un certain faisceau de propriétés, amorcés et interrompus par d'autres actes. Et, comme « les seules entités actives d'un système Unix sont les processus » [Tanenbaum 2003, 730], la réciproque est vraie : tout et seulement ce qui agit est processus.

La notion de processus est en informatique l'exact équivalent de ce que la notion d'acte est en ontologie. À proprement parler, les processus n'« existent » pas : ils opèrent. Ce sont des actes et non des objets. John Lions n'était donc pas si mal avisé, laissant la question aux philosophes, d'apparenter la question « qu'est-ce qu'un processus ? » à « qu'est-ce que la vie ? ».

2.2 Les fichiers comme objets

Qu'est-ce maintenant qu'un fichier ? Conformément au principe énoncé plus haut, observons la structure de données associée au concept de fichier dans Unix 6 afin de dégager ses propriétés essentielles.

```
5659: struct  inode
5660: {
5661: char i_flag;
5662: char i_count;    /* reference count */
5663: int  i_dev;      /* device where inode resides */
5664: int  i_number;   /* i number, 1-to-1 with device address */
5665:
5666: int  i_mode;
5667: char i_nlink;    /* directory entries */
5668: char i_uid;      /* owner */
5669: char i_gid;      /* group of owner */
5670: char i_size0;    /* most significant of size */
5671: char *i_size1;   /* least sig */
5672: int  i_addr[8]; /* device addresses constituting file */
5673: int  i_lastr;   /* last logical block read (for read-ahead)
     */
5674:
5675: } inode[NINODE];
```

Langage C

Un fichier est ici défini comme une entité identifiée par un certain numéro (`i_number`), appartenant à un propriétaire (`i_uid`) selon des droits d'accès déterminés (`i_mode`), située à une certaine adresse (`i_addr`) d'un certain périphérique (`i_dev`) et occupant une certaine longueur (`i_size0`). Cette structure de données est *introduite* et *éliminée* par les fonctions de création et d'effacement (`creat()` et `unlink()`, lignes 5781–5796 et 3510–3535), et manipulée par les fonctions d'ouverture et de fermeture (`open()` et `close()`, lignes 5765–5775 et 5846–5855), de lecture et d'écriture (`readi()` et `writei()`, lignes 6221–6320). La définition habituelle du fichier est donc cette fois légitime : « Un fichier Unix est une suite éventuellement vide d'octets contenant une information quelconque » [Tanenbaum 2003, 773]. Le fichier est un objet que certains actes peuvent créer, qui demeure stocké sur un périphérique, et que de nouveaux actes peuvent manipuler sous certaines conditions.

Le fichier est un objet, soit ; mais on peut aller bien plus loin en décrétant que tout objet est un fichier. Unix joua en effet un rôle majeur dans l'histoire des systèmes d'exploitation en généralisant ce concept avec la notion de « fichiers spéciaux » :

> Les fichiers spéciaux constituent la fonctionnalité la plus inhabituelle du système de fichiers d'Unix. Chaque périphérique d'entrée/sortie supporté par Unix est associé à au moins un fichier de ce type. Les fichiers spéciaux sont lus et écrits exactement comme les fichiers normaux sur disque, mais les demandes en lecture et écriture entraînent une activation du périphérique associé. [Ritchie & Thompson 1974, 367]

Tout objet, sous Unix, peut être considéré comme un fichier. Ce principe vaut évidemment pour les fichiers texte et les fichiers binaires que tout utilisateur de systèmes d'exploitation modernes manipule au quotidien — documents, images, films — mais également pour des objets moins triviaux. Il suffit de taper « `ls /dev/` », ou plus encore « `ls /proc/` » ou « `ls /sys/` » à l'invite de commande d'un système de la famille Unix pour constater avec surprise l'ampleur de ce qu'il considère comme fichiers. Le processeur, les disques durs, le clavier, l'écran, les informations relatives au processeur, la version du noyau ne sont que des fichiers. Même l'imprimante ! Le système d'exploitation en lui-même ignore tout de ce qu'est une imprimante : imprimer n'est pour lui rien d'autre qu'écrire dans le fichier `/dev/lp0`, dont il ignore tout, laissant aux pilotes de périphériques le soin de traduire le message dans l'idiome de ce que nous autres humains savons être une « imprimante ». De même, le scanner n'est pour lui qu'un fichier à lire ; la carte son, un fichier dans lequel on écrit ; le disque dur tout entier, un fichier que l'on lit et modifie.

> Traiter de cette manière les périphériques d'entrée/sortie présente un triple avantage : les entrées/sorties de fichiers et de périphériques sont aussi similaires que possible ; les noms de fichier et de périphérique possèdent la même syntaxe et la même signification, de sorte qu'un programme attendant comme paramètre un nom de fichier puisse recevoir un nom de périphérique ; enfin, les fichiers spéciaux sont sujets au même mécanisme de protection que les fichiers ordinaires. [Ritchie & Thompson 1974, 367]

On voit à quel point la notion de fichier qu'a adoptée Unix est éloignée de l'intuition familière que nous en avons. Pour Unix, non seulement tous les fichiers sont des objets manipulés par le système d'exploitation, mais tous les objets manipulés par le système d'exploitation sont des fichiers.

Si l'on entend par ontologie la *théorie de ce qui est*, le seul et unique engagement ontologique du système d'exploitation Unix et de tous ses descendants est donc que *tout ce qui agit est processus, et que tout ce qui existe est fichier*. Le couple que forment fichiers et processus ne tombe pas du ciel, mais d'une ontologie strictement composée d'actes et d'objets.

3 La catégorie des fichiers et des processus

Cette ontologie fonde une interaction. Une analyse du *shell* montrera que les propriétés caractéristiques de cette interaction sont exactement celles de la théorie mathématique des catégories.

Dans l'invite de commande standard d'Unix qu'est le *shell*, les processus ont le clavier pour entrée standard (`STDIN`), et l'écran pour sortie standard (`STDOUT`) ; c'est-à-dire que tout processus, en l'absence d'indications contraires, recevra ses arguments du « fichier spécial » qu'est le clavier (et son périphérique

caché qu'est l'être humain), et enverra ses résultats dans le fichier spécial qu'est l'écran.

Prenons pour exemple la commande Unix `cat`, dont la fonction est simplement de transcrire sur la sortie ce qu'elle reçoit sur l'entrée. Si l'utilisateur tape simplement « `cat` », le processus n'affichera rien en attendant de recevoir une entrée ; si l'utilisateur tape du texte puis appuie sur la touche d'entrée, le texte est simplement recopié :

```
invite_de_commande $ cat
Il y a de l'écho.
Il y a de l'écho.
invite_de_commande $
```

Par défaut, un processus comme celui qui précède est donc un acte

$$\texttt{STDIN} \xrightarrow{\texttt{cat}} \texttt{STDOUT}\ .$$

Le *shell* propose cependant des redirections permettant aux processus d'écrire de n'importe quel fichier vers n'importe quel fichier [Ritchie & Thompson 1974, 371]. Le caractère `<` permet de forcer l'utilisation d'un certain fichier comme entrée, le caractère `>` de forcer l'utilisation d'un certain fichier comme sortie. On peut ainsi demander à la commande `cat` de copier le fichier `entree.txt` dans le fichier `sortie.txt` :

```
invite_de_commande $ cat < entree.txt > sortie.txt
invite_de_commande $
```

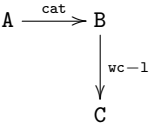
La commande peut échouer pour de nombreuses raisons, typiquement si le fichier d'entrée n'existe pas ou que le fichier de sortie soit protégé en écriture : la sortie d'erreur `STDERR`, par défaut l'écran, existe pour informer l'utilisateur de l'échec du remplissement de l'acte qu'il ambitionnait. Ainsi, au moins dans son intention, un processus quelconque peut être transformé en acte utilisant n'importe quel objet pour produire n'importe quel objet :

$$\texttt{entree.txt} \xrightarrow{\texttt{cat}} \texttt{sortie.txt}$$

Naturellement, le fichier de sortie d'un programme peut être utilisé comme entrée d'un nouveau programme, par une simple séquence d'instructions notée par l'utilisation du point-virgule. En utilisant la commande `cat` suivie de la commande `wc -l`, qui compte le nombre de lignes d'un fichier, on peut construire l'instruction suivante :

```
invite_de_commande $ cat < A > B ; wc -l < B > C
invite_de_commande $
```

Cette commande écrit le contenu du fichier `A` dans le fichier `B`, puis compte le nombre de lignes du fichier `B` et inscrit le résultat dans le fichier `C`. L'ensemble de la commande peut donc être vu comme un diagramme



Mais on peut préférer, par souci d'élégance et pour éviter de laisser éparpillées les traces d'étapes intermédiaires de calcul dont nous n'aurions plus besoin par la suite, ne créer les objets intermédiaires qu'à la volée au lieu de les écrire comme d'authentiques fichiers. Un tel procédé est caractéristique du style fonctionnel de programmation, celui du Lisp pur et du Haskell, par opposition au modèle impératif qui domine dans la plupart des langages de programmation — C, C++, Java, Perl, etc. Dennis Ritchie et Ken Thompson expliquent dans l'article de 1974 « The UNIX Time-Sharing System » que le concept de tube (*pipe*) permet d'économiser la création de fichiers temporaires [Ritchie & Thompson 1974, 371]. Les auteurs n'évoquent pas dans cet article la fonction plus générale du tube, qui est de faire communiquer des processus s'exécutant en parallèle. Mais selon les témoignages des créateurs d'Unix, telle n'était pas sa fonction originelle : absent de la première version d'Unix et du premier *Unix Programmer's Manual* [Ritchie & Thompson 1971], le concept fut introduit en 1972 sous la lourde insistance de Doug McIlroy, qui considérait les commandes comme des opérateurs binaires dont les opérandes de gauche et de droite étaient respectivement l'entrée et la sortie, ce qui permettait de les composer. Dans une note du 11 octobre 1964 datant du projet MULTICS, McIlroy comparait cette technique à un « tuyau d'arrosage » (*garden hose-screw*), métaphore qui marqua profondément les concepteurs d'Unix, Ken Thompson et Dennis Ritchie. Les explications qu'il leur livra, un après-midi sur le tableau noir, finirent par triompher de leurs réticences initiales [Ritchie 1984].

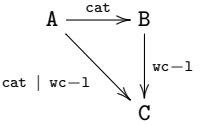
La commande précédente peut donc être réécrite en se passant de toute mention explicite du fichier `B`, qui n'est pas essentielle :

```
invite_de_commande $ cat < A | wc -l > C
invite_de_commande $
```

Une telle commande prend le contenu du fichier `A` en entrée, envoie le résultat dans un tube, c'est-à-dire un fichier temporaire, qui est immédiatement utilisé comme entrée du programme `wc -l`, dont le résultat est écrit dans le fichier `C`. L'utilisation de tubes, qui élimine les `B` d'une commande en branchant directement la sortie `B` d'une instruction sur l'entrée `B` de l'autre, correspond donc très précisément à la notion de coupure en théorie intuitionniste de la

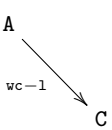
démonstration, qui correspond elle-même en mathématiques, comme l'observe Jean-Yves Girard[4], à la composition d'applications :

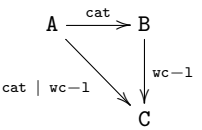
$$\frac{A \vdash B \qquad B \vdash C}{A \vdash C}\ \text{coupure}$$

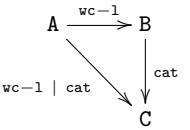


L'utilisation de tubes est associative et la commande `cat` peut y jouer le rôle de fonction identité. Les trois commandes suivantes sont en effet équivalentes :

```
invite_de_commande $ wc -l < A
invite_de_commande $ cat < A | wc -l
invite_de_commande $ wc -l < A | cat
```







Objets, actes, composition, identité, associativité : les interactions entre actes et objets dans l'ontologie de processus et de fichiers du système d'exploitation Unix présentent toutes les propriétés qu'un théoricien des catégories est en droit d'en attendre.

4 Thématisation et antithématisation

4.1 L'antithématisation en mathématiques

L'ontologie des actes et des objets est au cœur de l'analyse par Jean Cavaillès de la dynamique des mathématiques, qui permet d'éclairer certains concepts actuels de l'informatique.

Si les concepts d'acte et d'objet furent initialement empruntés à la phénoménologie, ils sont pourtant à considérer ici comme le produit d'une conscience anonyme ou polymorphe plutôt que comme celle d'un sujet mathématicien, fût-il idéalisé [Sinaceur 1990, 2581], [Sinaceur 1994, 99–100], [Schwartz 1998]. En concluant son ouvrage par l'évocation de « moments » d'une conscience mathématique qui se transforme et se révèle à elle-même dans l'histoire d'une

4. Au niveau logique qui correspond à la logique intuitionniste, « la règle de *Modus ponens*, ou plutôt la transitivité de l'implication, le *syllogisme* [qui sont des « cas particuliers de coupure », cf. p. 48], devient la *composition* des morphismes » [Girard 2006, 155].

« dialectique » [Cavaillès 2008b, 78] plutôt que comme une subjectivité personnelle et immuable, peut-être Cavaillès a-t-il voulu être à Husserl ce que Hegel avait été à Kant.

Les mathématiques appliquent certains actes à certains objets — par exemple l'addition à des nombres. Objets et actes sont pour ainsi dire « typés » : chaque acte est défini pour un certain type d'objet. Mais en complément au bagage d'actes fourni par le cours normal de la science, Cavaillès en a défini d'autres que l'on pourrait appeler, par analogie avec le lambda-calcul du second ordre [Girard 1972], [Reynolds 1974], « polymorphes », en ceci qu'ils peuvent s'appliquer à des objets de type quelconque. Ces actes polymorphes s'appellent paradigme et thématisation [Cavaillès 2008b, 27].

Étant donné un type et certains objets qui lui appartiennent, le passage au paradigme est la transformation des objets en objets plus généraux, et des actes en actes plus larges. À titre d'exemple, lorsque l'on généralise l'addition de nombres en addition de vecteurs, le concept de « nombre » a été élargi et l'addition revêtue d'une signification plus vaste, mais les objets sont restés des objets et les actes des actes.

La thématisation est l'acte polymorphe que Cavaillès définit comme opposée au paradigme. Elle fait abstraction des objets d'un certain type, considère les actes comme des objets, et détermine quels actes de niveau supérieur sont applicables à ces nouveaux objets. Tel est le cas lorsque des applications linéaires, d'abord actes effectués sur les objets particuliers que sont les vecteurs, deviennent à leur tour des objets, sous forme de matrices passibles d'actes de niveau supérieur tels que la mutiplication correspondant à la composition d'applications.

La dynamique de l'histoire des mathématiques progresse selon Cavaillès dans le sens du paradigme et de la thématisation, c'est-à-dire d'une abstraction croissante.

On peut cependant s'interroger sur l'asymétrie de l'opposition entre les deux concepts. La combinatoire des transformations possibles permet en effet théoriquement de définir (1) un geste transformant les objets en objets et les actes en actes — le paradigme ; (2) un geste transformant les actes en objets — la thématisation ; mais également (3) un geste *transformant les objets en actes*, et qui, s'il existait, serait le véritable inverse de la thématisation. Or lorsque Cavaillès envisage l'opération inverse de la thématisation, ce n'est que sur le mode du « principe de réductibilité » qu'il attribue à Husserl [Cavaillès 2008b, 50], selon lequel les constructions abstraites sont toujours en dernière instance convertibles en constructions concrètes, apparemment sur le seul mode du retour à l'origine. Redescendre l'arbre de l'abstraction ne pourrait être que revenir à des théories du passé, modèles particuliers — et réducteurs — des théories présentes. La « dialectique » des mathématiques, selon Cavaillès, évoluerait à sens unique vers l'abstraction.

Il ne serait pourtant pas monstrueux d'imaginer un processus inverse de la thématisation, disons d'« antithématisation », qui ne se réduise pas

à un simple retour aux objets passés, mais qui nous apprenne du nouveau sur ce que nous croyions être des objets atomiques. On trouverait ainsi en mathématiques l'analogue du constat enthousiaste de François Jacob selon lequel « même l'atome, le vieil irréductible, est devenu un système » [Jacob 1970, 344].

Le procédé existe. Les théoriciens des catégories ne font pas autre chose en considérant notre vieux singleton $\{a\}$ comme s'il était un acte, une transformation illustrée par le morphisme a de l'élément terminal $\mathbf{1}$ dans un ensemble A [Lawvere 1964, 1507] :

$$\mathbf{1} \xrightarrow{a} A$$

Et nous qui croyions naïvement que le singleton n'était un objet ! le voici devenu acte, prenant à contre-pied toute thématisation. On voit également le point, notre autre « vieil irréductible », devenir un morphisme de corps — quand ce n'est pas un foncteur, incarnation d'un acte opérant sur des actes. Il nous faut donc introduire, en regard du paradigme et de la thématisation, le concept d'antithématisation, qui eût peut-être paru monstrueux aux yeux de Cavaillès.

Contrairement à la thématisation, qui traite le complexe comme s'il était simple, l'antithématisation met au jour la complexité cachée du simple. Loin d'être condamnée au retour à des objets anciens et bien connus, l'antithématisation est parfois l'occasion de découvrir du nouveau dans ce que l'on croyait bien connaître. Les mathématiciens ont leur microscope.

4.2 L'antithématisation en informatique

Notre théorie des systèmes d'exploitation a les fichiers pour objets et les processus pour actes. Nous savons par ailleurs qu'il existe en mathématiques une transformation d'actes en objets — la thématisation — et une transformation d'objets en acte — l'antithématisation. Trouve-t-on l'équivalent dans l'ontologie des systèmes d'exploitation ?

On trouvera sans peine comment des processus peuvent, par thématisation, être transformés en fichiers. On en trouvera un premier exemple dans la *programmation*, écriture de programmes dont on attribue parfois l'idée moderne au « *First Draft of a Report on the EDVAC* » [von Neumann 1945], l'ordinateur électronique EDVAC, conçu de 1945 à 1949, permettant, contrairement à son prédécesseur l'ENIAC, conçu de 1943 à 1946, de charger des programmes archivés dans la mémoire ; mais l'idée de programmation était déjà inscrite au cœur de l'article fondateur de Turing [Turing 1936] : la machine universelle de Turing est une machine capable d'exécuter des programmes inscrits sur son propre ruban. Écrire un programme ou l'enregistrer en mémoire, que ce soit sous forme binaire ou textuelle, c'est faire d'un acte un objet ; car si le programme en cours d'exécution est un acte, son code source est un objet.

Par « sérialisation », un processus peut également être figé en un objet, ce qui permet par exemple sa transmission sur un autre support. Il en résulte

une ambiguïté dans la notion même de processus, selon qu'elle désigne l'acte en cours d'exécution ou l'objet qui l'incarne. John Lions montrait ainsi que le pseudo-parallélisme d'Unix 6 — c'est-à-dire le fait que l'absence de véritable parallélisme au niveau matériel soit compensée par une alternance dans l'exécution des processus, l'ordonnancement de leurs exécutions respectives étant laissée aux bons soins du système d'exploitation — nous contraignait à distinguer dans la notion de processus un niveau supérieur et un niveau inférieur :

> Au niveau supérieur, le « processus » est un concept important d'organisation pour décrire l'activité d'un système informatique pris dans son ensemble. [...] À ce niveau, les processus eux-mêmes sont considérés comme étant les entités actives du système [...] : les processus naissent, vivent et meurent ; ils existent en diverses quantités ; [...] ils peuvent interagir, coopérer, entrer en conflit, partager des ressources ; etc.
>
> Au niveau inférieur, les « processus » sont des entités inactives qui sont activées par des entités actives telles que le processeur. En permettant au processeur de passer fréquemment de l'exécution d'une image de processus à une autre, on peut créer l'impression que chacune des images de processus se développe continûment, ce qui mène à l'interprétation du niveau supérieur. [Lions 1996, 7-1b]

Certes, il ne s'agit là encore que d'un niveau assez rudimentaire de thématisation, dans la mesure où il n'y est question que d'interrompre et de reprendre l'exécution de processus. L'acte a beau devenir objet, cet objet n'est pas véritablement soumis à des actes de niveau supérieur. On n'obtiendrait donc guère plus que ce que Hegel appelait un « mauvais infini » : une répétition indéfinie à l'identique.

Mais le processus ainsi thématisé peut devenir objet pour des actes de niveau supérieur, donnant naissance au « bon infini » d'une réelle progression qualitative de la connaissance. Il est courant de voir des processus dont la fonction est d'écrire des programmes — et bien évidemment des processus écrivant des programmes à écrire des programmes. Les débogueurs sont des programmes capables de figer l'exécution d'un processus pour analyser, étape après étape, chacun de ses états instantanés, et par là son comportement général. L'utilisation d'un compilateur n'est autre qu'un processus analysant les propriétés d'un autre processus en en scrutant la forme objectifiée, à savoir le code source. La hiérarchie peut même encore être enrichie d'un niveau : lorsque Xavier Leroy a certifié en Coq le compilateur C CompCert, il a exécuté un programme dont la fonction était d'analyser à sa façon le texte d'un programme qui lui-même analysait d'une autre façon des programmes. Lorsqu'un assistant de preuve thématise un compilateur qui lui-même thématise des codes sources, la connaissance des actes de niveau inférieur progresse véritablement grâce à des niveaux supérieurs qui n'en sont pas le simple décalque. La thématisation

informatique n'est pas moins dialectique que celle que Cavaillès a vue dans les mathématiques.

Mais comme les mathématiques, la technologie des systèmes d'exploitation comprend également un procédé d'antithématisation, c'est-à-dire de transformation d'objets en actes. Telle est l'*exécution d'un programme*, lorsqu'un texte enregistré en mémoire est soudain considéré par le système — via un appel de type `exec()`, qui prend en argument un nom de fichier suivi des arguments qui seront passés au programme — non plus comme un objet existant pour lui-même et passible de certains actes tels que la lecture et l'écriture, mais comme une suite d'instructions que le système lui-même doit exécuter : un texte qui n'est pour lui pas tant à lire qu'à assumer. On voit une fois encore comme il est réducteur de définir le processus comme l'instance d'exécution d'un programme ; non seulement ce n'en est qu'un cas particulier, mais la notion de « programme » elle-même n'a de sens que définie à partir des concepts de fichier et de processus, et surtout de la transformation de l'un en l'autre.

Le concept d'antithématisation ne permet donc pas seulement de décrire un procédé que l'on trouve attesté dans l'histoire récente des mathématiques : il permet également de décrire un phénomène inscrit dans l'ontologie d'actes et d'objets qui structure les systèmes d'exploitation depuis Unix.

5 Conclusion

Les notions de processus et de fichier en ingénierie informatique ne sont donc pas les fruits du hasard et de la contingence. Elles sont la transposition d'une ontologie de l'acte et de l'objet dans l'ontologie des systèmes d'exploitation. Aussi n'est-il pas surprenant de constater que cette ontologie est accompagnée des deux procédés transformatoires dont témoigne l'histoire des mathématiques : le procédé de thématisation, transformation d'actes en objets de niveau supérieur, correspond à l'écriture ou à la manipulation de codes sources ; et le procédé d'antithématisation, qui transforme un objet en acte, correspond à l'exécution de programmes.

Peut-être cet exemple aura-t-il montré que les concepts des sciences de l'ingénieur, et par exemple de la théorie des systèmes d'exploitation, loin d'être condamnés à l'empirisme, peuvent relever d'une nécessité conceptuelle et être ancrés dans des concepts ontologiques fondamentaux que la philosophie a mission de dévoiler. Par ses concepts comme par ses procédés, la pratique informatique est pour la philosophie un objet aussi pur que les mathématiques.

Bibliographie

Bovet, Daniel P. & Cesato, Marco

2001 *Le Noyau Linux*, Paris : O'Reilly.

Cavaillès, Jean

2008a Remarques sur la formation de la théorie abstraite des ensembles, dans *Philosophie mathématique*, édité par Aron, R. & Martin, R., Visions des sciences, Paris : Hermann, 23–176.

2008b *Sur la Logique et la théorie de la science*, Bibliothèque des textes philosophiques, Paris : Vrin.

Gentzen, Gerhard

1955 *Recherches sur la déduction logique*, Philosophie de la matière, Paris : Presses Universitaires de France, 1re édition : "Untersuchungen über das logische Schliessen", *Math. Z.*, 39, 176–210 et 405–431. Traductions et commentaires par Robert Feys et Jean Ladrière.

Girard, Jean-Yves

1972 *Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur*, Thèse de doctorat, Université Paris VII.

2006 *Le Point aveugle. Cours de logique, I : Vers la Perfection*, Visions des sciences, Paris : Hermann.

Hegel, Georg Wilhelm Friedrich

2004 *Encyclopédie des sciences philosophiques, II. Philosophie de la nature*, Bibliothèque des textes philosophiques, Paris : Vrin.

Jacob, François

1970 *La Logique du vivant. Une histoire de l'hérédité*, Tel, Paris : Gallimard.

Lawvere, F. William

1964 An elementary theory of the category of sets, *Proceedings of the National Academy of Sciences of the United States of America*, 52(6), 1506–1511.

Lions, John

1996 *Lions' Commentary on UNIX 6th Edition with Source Code*, Charlottesville : Peer-to-Peer Communications.

Reynolds, John C.

1974 Towards a theory of type structure, dans *Lecture Notes in Computer Science*, Springer-Verlag, t. 19, 408–425.

Ritchie, Dennis M.

1984 The evolution of the Unix Time-sharing System, *Communications of the ACM*, 17, 365–375.

Ritchie, Dennis M. & Thompson, Ken

1971 Unix Programmer's Manual. `http://www.cs.bell-labs.com/who/dmr/1stEdman.html`.

1974 The UNIX Time Sharing System, *Communications of the ACM*, 17, 365–375.

Schwartz, Élisabeth

1998 Jean Cavaillès et la « philosophie du concept », *Philosophia Scientiæ*, 3(1), 79–97.

Sinaceur, Hourya

1990 Thématisation, dans *Encyclopédie philosophique universelle, Volume II, Les notions philosophiques*, Paris : Presses Universitaires de France.

1994 *Jean Cavaillès. Philosophie mathématique*, Philosophies, Paris : Presses Universitaires de France.

Tanenbaum, Andrew

2003 *Systèmes d'exploitation*, Paris : Pearson.

Tanenbaum, Andrew & Woodhull, Albert S.

1997 *Operating Systems. Design and Implementation*, Upper Saddle River : Prentice Hall.

Turing, Alan M.

1936 On computable numbers, with an application to the *Entscheidungsproblem*, *Proceedings of the London Mathematical Society*, 2(42).

Turner, Raymond & Eden, Amnon H.

2007 Towards a programming language ontology, dans *Computation, Information, Cognition—The Nexus and the Liminal*, édité par Dodig-Crnkovic, G. & Stuart, S., Cambridge : Cambridge Scholars Press, 147–159.

von Neumann, John

1945 First draft of a report on the EDVAC, Rap. tech., University of Pennsylvania, report prepared for U.S. Army Ordinance Department under Contract W-670-ORD-4926.

Vuillemin, Jules

1962 *La Philosophie de l'algèbre, tome I : Recherches sur quelques concepts et méthodes de l'Algèbre moderne*, Épiméthée, Paris : Presses Universitaires de France, réédition en 1993.